Gutorm Thomas Hogasen
University of Oslo
Institute of Mathematics
1988

About Multistate Fault-Trees.

Abstract.

The article will introduce the concept and analysis of the multistate fault tree. The discussed program Minipath will be a part of the Mustafa (MUlti STAte Faultree Analysis) Utilities. Minipath is a program designed to find the minimal path- and cut-sets of a multistate fault-tree. Among other things, multistate path and cuts can be used to find bounds for the availabilities and unavailabilities of a system [2][6].

I. Introduction.

This paper will treat the program Minipath. From an academic point of view, the interesting part of this program is how it generates minimal path and cuts sets. This is treated in detail in the paper. A section called "Using Minipath" is also included, with a run example and some additional information about the program. In this section, the environment Minipath will need to make a sensible output will be described as well. Please note that the purpose has been to develop an effective algorithm for analyzing the minimal path and cut sets in a multistate fault tree. The algorithm is not wrapped in a fancy environment with bells and whistles.

A. References quick reference.

McCullers III [1] is only used as inspiration for the definition of the multistate fault tree.

Funnemark and Natvig[2] shows some of the uses of path and cut sets of multistate systems.

Fussell and Vesely [3] contains the idea of the MOCUS algorithm.

Natvig [4] and Natvig [5] propose and discuss definitions for multistate systems. These produce the theoretical results that makes the definition and analysis of the multistate fault tree possible. The multistate fault tree proposed and analyzed in this article, is the Multistate Monotone System (MMS) as defined in Natvig [5].

Natvig[6] improves the bounds discussed in [2].

Willie[7] discusses the state of the art (1984) of binary fault tree analysis through minimal path and cuts. It also gives several algorithms for finding the minimal path and cut sets in a binary fault tree, and introduces the program FTAP. Many ideas from this article might be generalized to improve Minipath.

II. Notation.

The following notation will be used in this paper:

A. The indicator function I.

$$I(X) = \begin{cases} 1 & \text{if } X \text{ is true} \\ 0 & \text{if } X \text{ is false} \end{cases}$$

B. The structure function Φ.

$\Phi = \Phi(B)$ is the structure function. Φ describes the state of the system given the states of its components B.

C.Ordering of vectors.
We will order vectors as follows:

$A=(a_1 \ldots a_n) < B=(b_1 \ldots b_n)$
IFF $a_i \leq b_i$ $i=1 \ldots n$ and $a_i < b_i$ for at least one i

D.Replicated basic events.
Replicated basic events will be denoted RBE.

III.What is a multistate fault tree?

A.An example of a binary fault tree.
This is an example of a binary fault tree:

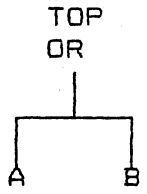


Figure 1.

Since this is a binary tree the nodes TOP,A and B will either function or be failed. (A=1, will mean that the event A occurs.) The state of node TOP is a binary function f of the nodes A and B. In figure 1 the function is OR, and may be expressed as

$$TOP = f(A,B) = 1 - (1-A)(1-B) .$$

The idea behind the multistate fault tree is: Why should we restrict ourselves to binary states for the nodes and the gate f?

In a multistate fault tree all nodes will have discrete states, and the gates will be discrete functions.

B.An example of a multistate fault tree.

Let TOP be the amount of power supplied by the sum of two generators A and B. A can supply 0,1,2,3 MW, while B can supply either 0 or 3. The maximum power we can transport is 5, and there is no need for the power if we get less than 2 MW.

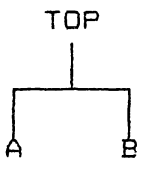


Figure 2.

$$TOP = f(A,B) = \min(5, I(A+B>1)*(A+B))$$

Note that B is a binary component, as failing means no power while functioning means that we get 3 MW.

TOP may here take the values 0,2,3,4,5.

C.Restrictions on the trees to by analyzed by Minipath.

The following restrictions apply to the trees to be analyzed by Minipath:

1) All gate functions f must be non-decreasing.

2) All nodes in the tree must have 0 as worst possible state.

3) All nodes must have a common m as best possible state.

Note that 1-3 imply that f(0)=0 and f(M)=m.
Natvig[4] discusses these assumptions, and shows their importance.
Note also that the tree in figure 2 can not be analyzed by Minipath, as node A and B have 3 as best possible state, while node TOP can attain 5.

D.The dual tree.

The dual tree is found by replacing all states s by $s_D=m-s$ and all the gate functions f(S) by $f_D(S)=m-f(M-S)=m-f(S_D)$. m is here the common best possible state for a node, while $M = (m,m, \ldots ,m)$.

Note that if a tree T can be analyzed by Minipath, the dual tree T_D may be as well. This is true due to the following remarks:

1) If all gate functions f(S) are non-decreasing, $m - f(M-S) = f_D(S)$ is non-decreasing as well.

2)
$$f_D(0) = m - f(M-0) = m - m = 0$$
$$f_D(M) = m - f(M-M) = m - 0 = m$$

IV.Definitions (Based upon McCullers III[1] and Natvig[4])

A multistate fault tree is a directed graph F(E,L) where E is the set of events and L is the set of links. An event $e \in E$ is a pair e=(v,f) where $v \in V$ is the event vertex, and f is the event function. (In the binary situation, f could be And, Or, Xor etc.) In our analysis, we will restrict ourselves to systems where all f are non-decreasing, f(0)=0 and f(M)=m.

To each event is associated the event states. e will have possible states as the elements of the discrete set S_e. (In the binary situation, S_e will always be {0,1} for all e) Events are connected by links $l_{i,j} \in L$. $l_{i,j}$ transmits the output from event i to the input of event j. The outdegree

of e_i is the number of j such that $l_{i,j} \in L$. The indegree of e_j is the number i such that $l_{i,j} \in L$.

An event with indegree 0, is called a basic event. A basic event with outdegree > 1 is called a replicated basic event (RBE). A top event is an event with outdegree 0.

Let T be the set of all basic events. Given the state of T, the state of all other events will be uniquely given by F. Let $\Phi(B)=\Phi$ be the structure function of the fault-tree.

A minimal path vector for Φ on level k is a B such that

$$\Phi(B) \geq k \text{ and } \Phi(A) < k \text{ for all } A < B .$$

It is important to realize that a path can be minimal on several levels (as stated in Natvig[4]). This is an unknown problem in the binary fault tree analysis, but will cause a great deal of trouble in the multistate case.

A minimal cut vector for Φ on level 1 is a B such that

$$\Phi(B) < 1 \text{ and } \Phi(C) \geq 1 \text{ for all } C > B .$$

If B is the set of all basic events, the minimal path/cut set will be a global set. We will also talk about minimal local paths: A local minimal path for a node e on level k will be expressed by the states of the nodes right below e.

V. Remarks on the relation between minimal path- and cut-sets and the dual tree.

From the minimal path sets for the tree T, the minimal cut sets for the dual tree T_D may be found, according to the following statement from Funnemark and Natvig[2]:

A is a minimal path on level k for the tree T
<=> (M-A) is a minimal cut set on level m-k+1 for T_D

Proof: Let $A_D = (M-A)$. Note that $(A_D)_D = A$.

It is enough to show the result for a specific node in the tree, as it then will be true for all nodes in the tree, and in particular for the top event. Let the gate function for the node be f, and A a minimal (global or local) path. Let B < A.

$f(A) \geq k$ since A is a path on level k for f
<=> $f_D(A_D) = m - f((A_D)_D) = m - f(A) \leq m-k$
<=> $f_D(A_D) < m-k+1$ (1)

The same argument gives:

$f(B) < k$ since A > B
<=> $f_D(B_D) \geq m-k+1$ (2)

The definition of a minimal cut set together with (1) and (2) shows that A_D is a minimal cut set on level m-k+1 for f_D.

This result will be used later to find the minimal cuts in a multistate fault tree: When the algorithm for finding the minimal path sets is ready, we will simply find the dual tree T_D and apply the algorithm on T_D.

VI. The main idea of the algorithm.

The essence of the algorithm is similar to the one in MOCUS[3] developed for the binary case.

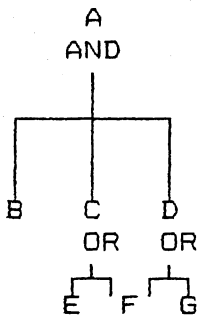
> [It is well known that better algorithms exists in the binary situation (MSDOWN, MSUP in Willie[7]). These methods are essentially based on two ideas:
>
> The first is modular decomposition of the tree. It is clear that this would improve the efficiency of Minipath as well.
>
> The second is that given (non-minimal) path sets, we can find the minimal paths by finding the dual set of the dual set of the paths. This is based on the definition Willie[7] uses for dual sets, ensuring minimality. Willie[7] sites (page 43), an algorithm for finding the dual of a given set. A corresponding algorithm does not yet exist for the multistate case.]

Some enhancements are possible in the algorithm proposed in this paper, and will be discussed under the section "Bugs".

We will from now only talk about the minimal path set analysis. The minimal cut sets may as seen above be found by finding the minimal path sets for the dual tree.

Let us look at the following binary example:

Figure 3.



Top event: A.
Basic events: B,E,F,G.
Replicated basic event: F.

Possible states: (0,1) for all events.

We (and a computer) can easily see that

i) (B=1, C=1, D=1) is a (the) minimal local path for node A on level 1.

However C and D are not basic events. But

ii) (E=0,F=1) and (E=1,F=0) are minimal paths for C on level 1.

iii) (F=1,G=0) and (F=0,G=1) are minimal paths for D on level 1.

Combining i) and ii), this suggests to use

(B=1,E=0,F=1,D=1),
(B=1,E=1,F=0,D=1) as minimal paths for A on level 1.

This is also the principal idea of the algorithm:

1) Find the minimal local paths for all nodes and all levels.
2) Substitute X=a with the minimal local paths for node X on level a, for all nodes X in the tree. Repeat this until the top event is expressed as minimal paths consisting of only basic events.

As long as the tree contains no RBE, this algorithm works well and is efficient. Replicated basic events will however cause problems: If this algorithm is applied to the tree in figure 3, we would end up with

(B=1,E=0,F=1,F=0,G=1) as one minimal path for A on level 1.

One will note that F has to different values in this "path".

In the general case, the situation is this:

Figure 4.

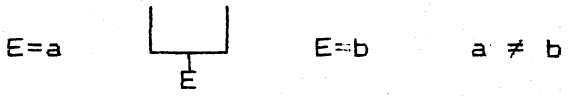


Figure 4 illustrates that the left hand node over E dictates E=a in order to make a minimal path, while the right hand node wants E=b. The problem is what to do in such situations.

There are three possible choices, illustrated by the example above as:

1) Ignore the found path.
2) Use the smallest of the suggested values for E.
3) Use the largest of the suggested values for E.

Solution 1) can not be applied, as we might loose certain minimal paths.

> [Look at the example in figure 3. Make the following changes:
>
> C(E,F) = I(E+F>1)
> D(F,G) = min(1,F,G),
> F ∈ {0,1,2}
>
> Now analyze the new tree with the method described above, ignoring the paths causing trouble. You will then loose a minimal path.]

Solution 2) can not be applied either. Assume a<b in figure 4. If we try to use E=a, the left hand side will be ok, while the right hand side will collapse: E=b was the least value accepted to make a path on the right hand side. Since all local paths are minimal, they will make <u>all</u> nodes drop a level on the right hand side of E.

As a conclusion, we must use the largest of the two possible values. But this procedure may generate paths that are not minimal. In the example of figure 3, the paths

> (B=1,E=0,F=1,G=1) and (B=1,E=1,F=1,G=0)

will be generated without being minimal. (In the first (second) path, we could let the state of component G (E) drop to 0 without influencing the state (1) of the top event.)

The procedure will be, in a collision situation, to use the best state of the RBE and check the generated paths to see if they are minimal.

VII. <u>Implementation and portability.</u>

The program is written in Turbo Pascal under MsDos. It can therefore be run on any IBM PC clone. The maximum size of the tree that can be analyzed is dependent on the available memory (Min. 64k). To the extent it was possible, only standard Pascal was used, meaning that Minipath easily may be ported to a larger and much more powerful computer. The very last part of the algorithm (sorting the path and cut sets found), is the most time consuming, and could be done much more efficient on a machine with more memory.

In order to be able to analyze fairly large trees on a PC, the program will use much disk space during the execution. This slows down the program, but it was too much of a temptation to make a PC program being able to analyze fairly big trees. A binary tree with 76 basic events, was successfully analyzed and produced 152 minimal cuts and 432 minimal paths. However, the execution time was about two hours.

In the following we will go into detail of the two phases:

A) Generating potential minimal paths.
B) Eliminating the non-minimal paths.

A.Generating potential minimal paths.

At this point the program will already have found all minimal local paths, a trivial problem. (Well, at least it is solved in a trivial way; For each event e with nodes n_1 ... n_m right below, all combinations of the states of n_1 ... n_m will be run through. If decreasing the state of any of the subnodes make e drop one or more states, we have found a minimal path, and will record it according to what level it dropped from. Finding the local minimal paths with this method is quite fast for ordinary trees. However, if one has a tree with nodes having hundreds of states and hundreds of direct subnodes, the process may be painful)

The problem now is to find a good way of linking the information about local minimal paths into global minimal paths. This could of course be done by expanding all occurrences of A=a in the local minimal paths with the minimal paths for A on level a. Unfortunately, this method would literally eat up the computer's memory, since it requires that all minimal paths on all levels must be kept in memory, and a considerable amount of copying has to be done by the program. A more refined method is therefore used, using less memory but more time.

Our problem can be translated into this one: Each local minimal path can be thought of as an array of integers (giving the states of the nodes below). Each such integer i can be expanded into several other integer arrays: The local minimal paths for the nodes below. The problem is to expand small arrays of integers (local paths) into larger arrays (global paths).

Example:

Let a_1,b_1,c_1 represent a local minimal path.
Let a_1 be expanded into x1,y1,z1 or x_2,y_2,z_2
Let b_1 be expanded into k_1,l_1
or k_2,l_2
or k_3,l_3
Let c_1 be a basic event.

What we want to generate is

$x_1,y_1,z_1.k_1,l_1,c_1$
$x_2,y_2,z_2.k_1,l_1,c_1$
$x_1,y_1,z_1.k_2,l_2,c_1$
$x_2,y_2,z_2.k_2,l_2,c_1$
$x_1,y_1,z_1.k_3,l_3,c_1$
$x_2,y_2,z_2.k_3,l_3,c_1$

This will be done by using two help arrays, A and B. The elements of A will be basic events, while those of B will be nodes to be expanded. When B is empty, a potential minimal path is generated.

We will start out with

A: B: a_1, b_1, c_1

Since c_1 can not be expanded, it will be moved to A.

A:c_1 B: a_1, b_1

Now b_1 will be expanded into three possibilities.

First A:c_1 B: a_1, k_1, l_1
next A:c_1 B: a_1, k_2, l_2
and so on

Since k_1 and l_1 are basic events, they will be moved directly over to A. The program will now process a_1 in the same manner. When a_1 has been expanded, we will return to expand b_1 into k_2 and l_2.
The implementation of the idea above is done in the following way:

```
1: Procedure Track(a,b);
2:    If B empty, check A, RETURN
3:    help:= B(b)
4:    if B(b) basic event:
5:          Move B(b) over to A(a+1)
6:          Track(a+1,b-1)
7:          B(b):= help
8:          RETURN
9:    if B(b) can be expanded:
10:         for all arrays B(b) can be expanded into do
11:            Expand B(b) .
12:            newb:=address of the 1. unexpanded value in B
13:            track(a,newb)
14:         goto 11:
15:         B(b):=help
16:         RETURN
```

Comments:

The variables a and b are local to the procedure Track, and will not change when various subroutines are called from Track.

The arrays A and B are global, and may be changed when calling other subroutines. However, Track will make sure not to destroy their interesting parts: The elements 1...b contain the local global paths we are to expand, and must not be destroyed by calling Track. (This may happen when Track(*,b-1) is called, where * may be any value of a. If element b-1 of B is expanded, element b will be replaced by something else, thus destroying B(b).) To avoid this, B(b) will be saved (line 3) before calling Track, and restored after Track has returned (line 7 or 15).

We are not concerned about what happens with A, as a call on Track only will destroy elements A(a+1) ... A(n), and our path at the moment is contained in A(1) to A(a).

1:The (recursive) procedure will use the global arrays A and B.(Not to be confounded with the integers a and b, that design the position of an element of A and B) A(1)...A(a) are basic events, while B(1)...B(b) are elements to be expanded. A potential minimal path will be contained in A(1)...A(a) if B is empty. A will be checked for minimality if necessary, that is if a certain flag is set. (See "Eliminating the Non-minimal paths".)

2:A is only a potential minimal path. If it indeed is minimal, it will be written to disk. In all cases we will return (withdraw) to the previous level of the recursion, and continue the analysis.

3:Expanding B(b) will destroy this element. When withdrawing we will need to restore it.

5:B(b) is a basic event, and is moved to the first free place of A.

6:Continuing the work, with our arrays updated in 5:

7:Restoring B(b) as it was before withdrawing.

10: The loop from 11: including 13: will be performed once for each expansion of B(b). If B(b) can be expanded into x_1,y_1,z_1 and x_2,y_2,z_2 it will be done twice.

11: Expand is a procedure that replaces B(b) with an array (a minimal path), thus increasing the length of B. With the example from 10:, it would be increased from B(1)...B(b-1) as it was, to B(b)=x1, B(b+1)=y_1 and B(b+2)=z_1.

12: The example over would have given us newb=b+2

13: Track on: No change in A, B is updated.

15: Restoring B(b) as it was before withdrawing.

As one can see, there is no copying of paths, except for the moving of elements from B to A. This means that once a minimal path is generated, it will be forgotten, and not hang around filling up memory.

Alas there is a BIG drawback: If an a element occurs in several paths, it will be expanded several times; we could have stored the expanded a and recalled it the second time. However, it would consume much more memory. Clearly, a good deal of improving could be made here, especially for modules: Identifying elements that will be expanded many times and storing them away.

B.Eliminating the non-minimal paths.

Unfortunately, there is still a minor problem. When we have expanded all the elements of B, A will not necessary contain a minimal path, as seen from the examples. The problem is now: When is it necessary to check the generated array for minimality, and how can it be done?

Our problem has two causes: The replicated basic events, and the fact that a local minimal path on level k may also be minimal on level (k+1).

If our generated path contained different values for a replicated event (remember we choose the largest one), we might be in trouble and a check for minimality is required. Furthermore, if we expanded the value k with a path being minimal on both levels k and k+1, we must control the minimality. The program will only check for minimality if one of the events above occur.

An example illustrating the necessity of checking for minimality is as follows: Consider a node N with the nodes A and B below it, and with gate function A+B. A minimal path for level 4 would then be (A=2,B=2). During our analysis, we will replace this minimal local path with minimal global paths. Now suppose one of the minimal paths for B on level 2, also is minimal for B on level 3. The global path returned would then yield B=3. Alas, the local path (A=2,B=3) is clearly not minimal for N on level 4 if 1 is a possible state for node A.

The test for minimality is done straight forward: Minipath tries to reduce the state of all the basic events (one at a time), and if this does not make the state of the top level drop, a false path has been introduced. The reader may feel that it is unnecessary to check all the basic events, and is perfectly right. Alas, finding a good way of picking the crucial elements is not trivial, and the program makes no attempt in being clever at this. However, it is clear that modularisation will be a key word in such an analysis.

The problems causing non-minimal paths to be generated, also makes it possible to find the same path several times. To eliminate the doubles is purely a sorting problem: Once the paths are all sorted, it is easy to find the doubles. The program outputs files of minimal path and cuts for each possible level of the top event, and sorts them with a standard sort utility supplied with the operating system. Finally, if two lines following each other are equal, the latter will be deleted.

C.Bugs.

1. Modularisation is not used. The problem of recognizing modules is however identical to the one for binary fault trees, and good algorithms exists here. (Willie[7]).

2. It is well known that for certain (sub)systems, the path analysis is easier to perform by first finding the systems dual tree, and then finding its minimal cuts. This point is however totally missed by the program.

3. The gate functions must be entered as tables. This is quite painful for nodes with many states and many direct subnodes. It should be possible to choose between entering a formula or a table. The program relies on having a table describing each gate function, but should be able to produce this table from a function.

4. When the program searches for a minimal path on level k, and returns one on level (k+1), it will set a flag and then check for minimality later, when all paths are expanded. (Remember this may happen, as a path can be minimal both on level k and (k+1)). It is in the general case difficult to predict what consequences this might have for the top event, but it should in some cases be possible to realize that the track worked on impossibly can produce a minimal path, and therefore abort the search immediately.

5. The program never checks for minimality before a path is completely expanded. Time could certainly be saved by doing some minimalization on earlier stages of the expansion.

6. Recognizing duplicate minimal paths is a sorting problem. It is rather clumsily solved by using the standard sort utility (SORT) which comes with MsDos. Alas, SORT is a terrible program. In fact, the execution of SORT is one of the most time consuming parts of Minipath. However, since any sorting program could be used, the problem is easy to reduce.

VIII. Using Minipath.

A. Strategy of the program.

Minipath needs input from a file describing the structure of the tree to be analyzed. The easiest way to make such a file is to use the program Cretre.

When the input file is generated, you should start up Minipath with the command Mustafa. Minpath will produce a batch file as result (&foo.bat), and Mustafa will make sure you run this batch file.

The batch file will then manipulate a number of files created by Minipath, in order to get rid of some duplicate path and cut sets. Finally, the batch file will create the result file you ask for.

Note that files will be created on the directory you currently reside on. These files are called &p? and &c? where p (c) stands for path (cut), while ? will be replaced by all possible states for the top event. These files will

be erased automatically. This means that the program must be able to create files: It will crash if the disk is full or write protected.

B.Files needed.

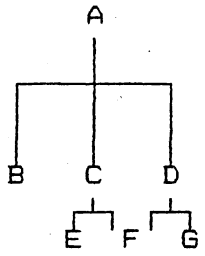
These are the files needed:

Cretre.com:	A program to generate the tree.
Mustafa.bat:	A batch file used to start other programs.
Minipath.com:	The program finding the paths and cuts
Unique.com:	A program that deletes one of two identical lines in a file.
Sort.exe:	The (Ms)Dos sort utility

A number of other files will be created as well during the execution, all having names starting with '&' .

C.An example

We will analyze the following tree:



Top event: A.
Basic events: B,E,F,G.
Replicated basic event: F.

Let a(B,C,D) be the state of A given the state of the nodes below, the structure function of A. Let us use the following structure functions:

a: max(B,C,D) but never more than 5
c: E+F but never more than 5
d: the integer value of (F+G)/2

Let the basic events B,E and G all have states in [0,5]
Let F have states in [0,1,2,4,5]

C can then have states in [0,1,2,4,5]
D can have states in [0,1,2,3,4,5]
A can have states in [0,1,2,3,4,5]

Note that all components have 0 as worst possible state and 5 as best, and that all gate functions are non-decreasing.

The nodes A-G will be given the numbers 1-7.

Run Example (user given commands in bold face)

C>cretre

```
This program creates a multistate fault-tree.
Do you want to modify a tree saved on a file? n
Number of the top node? 1
Give the first node below node 1 (Exit with 0):2
Next: 3
Next: 4
Next: 0
Give the first state of node 1 :0
Next state (exit with negative state): 1
Next state (exit with negative state): 2
Next state (exit with negative state): 3
Next state (exit with negative state): 4
Next state (exit with negative state): 5
Next state (exit with negative state): -1

Next node to be entered? ( 0 to exit) 2
Give the first node below node 2 (Exit with 0):0
Give the first state of node 2 :0
Next state (exit with negative state): 5
Next state (exit with negative state): -1

Next node to be entered? ( 0 to exit)  <enter all nodes>

Next node to be entered? ( 0 to exit) 0
Ok, all nodes given. Now the structure functions...

Next node to be entered? ( 0 to exit) 1

(Structure function for node 1 )
Node states below are
Node 4 with state 5
Node 3 with state 5
Node 2 with state 5
     Gives state: 5

<Give all the structure functions>
<The last will be: >
(Structure function for node 4 )
Node states below are
Node 7 with state 5
Node 6 with state 0
     Gives state: 2

(Structure function for node 4 )
Node states below are
Node 7 with state 0
Node 6 with state 0
     Gives state: 0

Next node to be entered? ( 0 to exit) 0

Do you want me to check that all best and worst states are
the same? y
Give the worst state for a node to be in: 0
Give the best state for a node to be in: 5
Check done, all errors reported.
On what file do you want the tree stored? paper.tre
```

C>

The tree is now created in the file paper.tre .
The command
C>mustafa
will now produce the following result on a file:

```
   Paths for level 5
    7    6    5    2
 --------------------
    0    0    0    5
    0    0    5    0
    0    5    0    0

   Paths for level 4
    7    6    5    2
 --------------------
    0    0    0    5
    0    0    5    0
    0    4    0    0

   Paths for level 3
    7    6    5    2
 --------------------
    0    0    0    5
    0    0    5    0
    0    4    0    0
    5    1    0    0

   Paths for level 2
   7    6    5    2
 --------------------
    0    0    0    5
    0    0    5    0
    0    2    0    0
    5    0    0    0

   Paths for level 1
    7    6    5    2
 --------------------
    0    0    0    5
    0    0    5    0
    0    1    0    0
    5    0    0    0

   Cuts for level 5
    7    6    5    2
 --------------------
    5    4    0    0

   Cuts for level 4
    7    6    5    2
 --------------------
    5    2    0    0
```

Cuts for level 3

7	6	5	2
0	2	0	0
5	0	0	0

Cuts for level 2

7	6	5	2
0	1	0	0

Cuts for level 1

7	6	5	2
0	0	0	0

The matrix

Paths for level 5

7	6	5	2
0	0	0	5
0	0	5	0
0	5	0	0

will mean that component B in state 5 and the others in state 0 is the first minimal path for the tree on level 5, while the two other minimal path are given by the two last lines in the matrix.

IX.Conclusion.

The first step in multistate fault tree analysis is done. Unfortunately, the path is still long before a really decent program is ready, a program being able to challenge the best binary fault tree programs made.

X.Acknowledgment

I am very grateful to professor Bent Natvig for help, encouragement and interesting discussions.

XI. References:

[1] McCullers III, W. T. Probabilistic Analysis of Fault Trees Using Pivotal Decomposition, Ph.D. Thesis, Naval Postgraduate School, Monterey, California, June 1985.

[2] Funnemark, E. and Natvig, B. Bounds for the Availabilities in a Fixed Time Interval for Multistate Monotone Systems; Adv. Appl Prob 17, 638-665 (1985).

[3] Fussel, J. B. and Vesely, W. E. A methodology for Obtaining Cut Sets; American Nuclear Society Transactions, Vol. 15, no. 1, pp 262-263, June 1972.

[4] Natvig, B. Two suggestions on how to define a Multistate Coherent System. Adv. Appl Prob 14, 434-455 (1982).

[5] Natvig, B. Multistate Coherent Systems. In Encyclopedia of Statistical Sciences, Vol. 5, ed. N. L. Johnson and S. Kotz. Wiley, New York, 732-735 (1985)

[6] Natvig,B. Improved Bounds for the Availabilities in a Fixed Time Interval for Multistate Monotone Systems; Adv. Appl Prob 18, 577-579 (1986).

[7] Willie, R.R. Computer Aided Fault Tree Analysis, 1984 Operation Research Center, University of California, Berkeley, ORC 78-14